Code Profiling Report

1. **Where the code is spending more time**

   The majority of time is spent in the 'gol' function and its children functions, since it is the function that runs the game itself. From this one there are a lot of calls to 'cell_alive', which is executed every time that there is a need to check if a cell is going to be alive in the next generation. Since it is checking for each cell in the board for each generation, it makes sense to be the most time consuming function of the code. After this, the 'print_cells' is the other children function from 'gol' that spends a reasonable amount of time, and it is called one time per generation in order to print the board state.

   In the case of the other 'main' function children, both 'init_pattern' and 'selectPattern' are executed only when the game begins or restarts, so they nearly do not consume any time in comparison with the others. The 'main' function itself spends some time due to the initialization of the program and the exiting of the game.

   The profiling result can be checked at the end of this file, in the Annex.

2. **Memory allocation statistics from your code**

   Most of the memory being allocated is due to the use of ncurses library. Without visualisation it would be possible to use just a fraction of what is being allocated now from the heap for the program to be working. In our case, the program needed 66 KiB of allocated memory at the peak of memory consumption.

   It has been found that the type of initial pattern or time the program runs does not affect the amount of memory being allocated, not even the peak amount of allocated memory. Therefore one example of memory allocation needs to be presented. In Figure 2 can be seen a graph of memory consumption of the Game of life project. The program ran for 111 generations for 2 minutes (5 seconds setting up, cca 5 seconds ending the game), with initial configuration 'Die hard'. Since the graph consists of a large amount of information, not everything might be legible. Important information can be found in Figure 1, which represents the peak of memory consumption for the time the program was running.

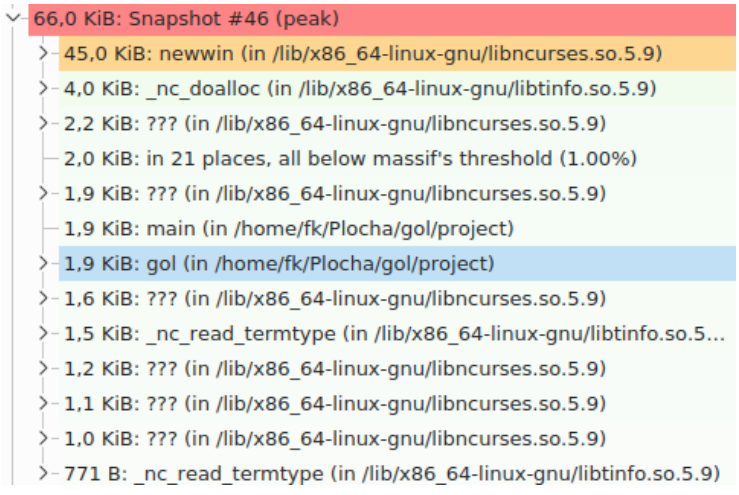

Figure 1: Peak of memory consumption of game of life program

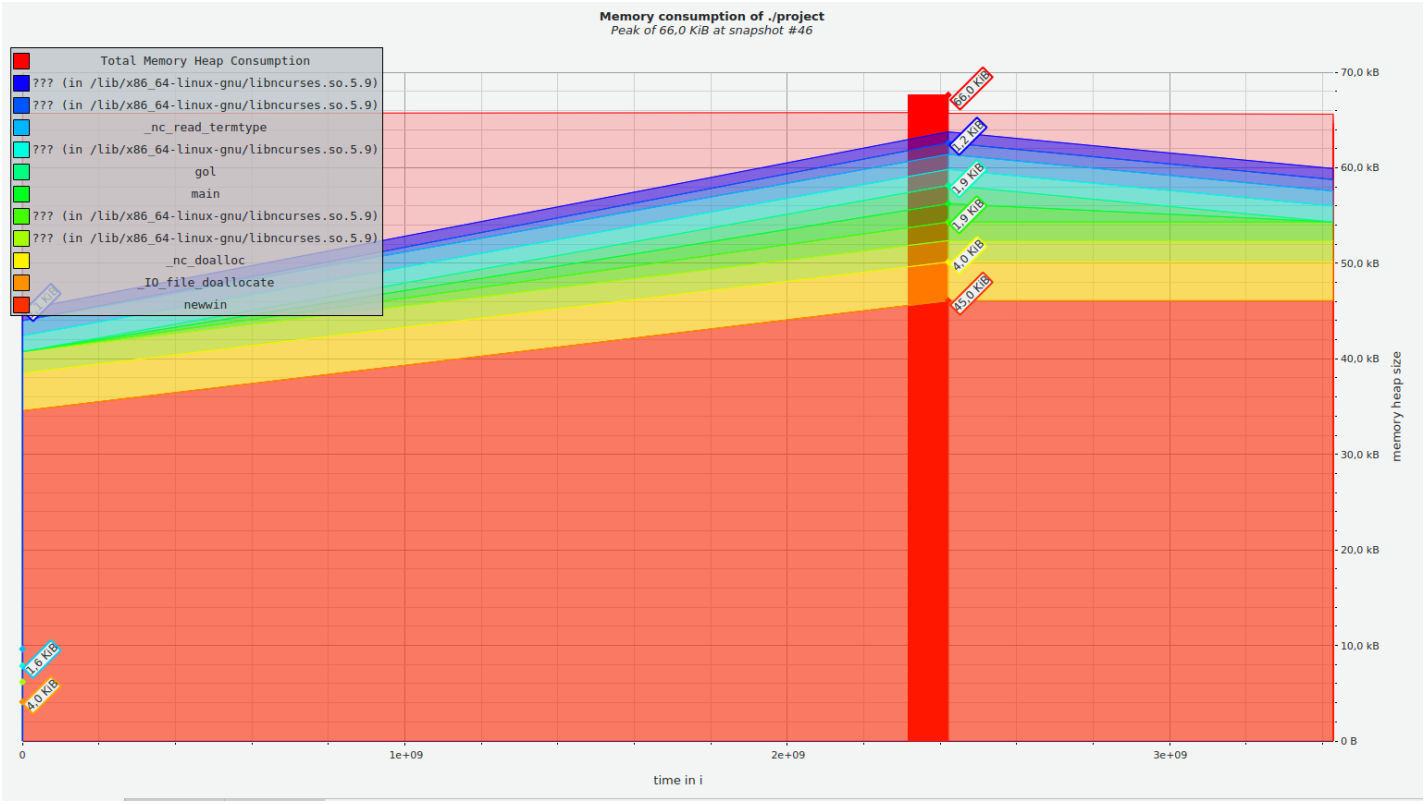

Figure 2: Graph of memory consumption of game of life program

3. **Check of memory leaks in your code.**

After running Memcheck from Valgrind to see if there are any memory leaks in the program, it shows that it does not have any:

==32403== Memcheck, a memory error detector
==32403== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==32403== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==32403== Command: ./project
==32403==
==32403==
==32403== HEAP SUMMARY:
==32403== in use at exit: 49,155 bytes in 93 blocks
==32403== total heap usage: 144 allocs, 51 frees, 61,055 bytes allocated
==32403==
==32403== LEAK SUMMARY:
==32403== definitely lost: 0 bytes in 0 blocks
==32403== indirectly lost: 0 bytes in 0 blocks
==32403== possibly lost: 0 bytes in 0 blocks
==32403== still reachable: 49,155 bytes in 93 blocks
==32403== suppressed: 0 bytes in 0 blocks
==32403== Reachable blocks (those to which a pointer was found) are not shown.
==32403== To see them, rerun with: --leak-check=full --show-leak-kinds=all
==32403==
==32403== For counts of detected and suppressed errors, rerun with: -v
==32403== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

Annex:

Time profiling report output:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self   total
 time   seconds   seconds  calls  ms/call  ms/call  name
 50.03       0.07   0.07  1469820       0.00   0.00  cell_alive
 26.94       0.10   0.04   130    0.27   0.27  print_cells
 23.09       0.13   0.03          1      30.02   130.08  gol
  0.00       0.13   0.00          1      0.00   0.00  init_pattern
  0.00       0.13   0.00          1      0.00   0.00  selectPattern

 %           the percentage of the total running time of the
time         program used by this function.

cumulative a running sum of the number of seconds accounted
 seconds   for by this function and those listed above it.

 self        the number of seconds accounted for by this
seconds      function alone.  This is the major sort for this
             listing.

calls        the number of times this function was invoked, if
             this function is profiled, else blank.

 self        the average number of milliseconds spent in this
ms/call      function per call, if this function is profiled,
       else blank.

 total       the average number of milliseconds spent in this
ms/call      function and its descendents per call, if this
       function is profiled, else blank.

name         the name of the function.  This is the minor sort
             for this listing. The index shows the location of
       the function in the gprof listing. If the index is
       in parenthesis it shows where it would appear in
       the gprof listing if it were to be printed.

Copyright (C) 2012-2018 Free Software Foundation, Inc.

Copying and distribution of this file, with or without modification,
are permitted in any medium without royalty provided the copyright
notice and this notice are preserved.

             Call graph (explanation follows)


granularity: each sample hit covers 2 byte(s) for 7.69% of 0.13 seconds

index % time  self  children      called       name
             0.03   0.10          1/1           main [2]
[1]    100.0  0.03   0.10          1      gol [1]
             0.07   0.00 1469820/1469820       cell_alive [3]
```

```
             0.04   0.00   130/130      print_cells [4]
-----------------------------------------------
                                        <spontaneous>
[2]    100.0  0.00   0.13               main [2]
             0.03   0.10         1/1           gol [1]
             0.00   0.00         1/1           selectPattern [6]
             0.00   0.00         1/1           init_pattern [5]
-----------------------------------------------
             0.07   0.00 1469820/1469820      gol [1]
[3]    50.0   0.07   0.00 1469820        cell_alive [3]
-----------------------------------------------
             0.04   0.00   130/130      gol [1]
[4]    26.9   0.04   0.00   130          print_cells [4]
-----------------------------------------------
             0.00   0.00         1/1           main [2]
[5]    0.0    0.00   0.00         1      init_pattern [5]
-----------------------------------------------
             0.00   0.00         1/1           main [2]
[6]    0.0    0.00   0.00         1      selectPattern [6]
-----------------------------------------------
```

This table describes the call tree of the program, and was sorted by the total amount of time spent in each function and its children.

Each entry in this table consists of several lines. The line with the index number at the left hand margin lists the current function. The lines above it list the functions that called this function, and the lines below it list the functions this one called. This line lists:

index A unique number given to each element of the table. Index numbers are sorted numerically. The index number is printed next to every function name so it is easier to look up where the function is in the table.

% time This is the percentage of the `total' time that was spent in this function and its children. Note that due to different viewpoints, functions excluded by options, etc, these numbers will NOT add up to 100%.

self This is the total amount of time spent in this function.

children This is the total amount of time propagated into this function by its children.

called This is the number of times the function was called. If the function called itself recursively, the number only includes non-recursive calls, and is followed by a `+' and the number of recursive calls.

name The name of the current function. The index number is printed after it. If the function is a member of a cycle, the cycle number is printed between the function's name and the index number.

For the function's parents, the fields have the following meanings:

self This is the amount of time that was propagated directly from the function into this parent.

children   This is the amount of time that was propagated from the function's children into this parent.

called   This is the number of times this parent called the function `/' the total number of times the function was called.  Recursive calls to the function are not included in the number after the `/'.

name   This is the name of the parent. The parent's index number is printed after it. If the parent is a member of a cycle, the cycle number is printed between the name and the index number.

If the parents of the function cannot be determined, the word `<spontaneous>' is printed in the `name' field, and all the other fields are blank.

For the function's children, the fields have the following meanings:

self   This is the amount of time that was propagated directly from the child into the function.

children   This is the amount of time that was propagated from the child's children to the function.

called   This is the number of times the function called this child `/' the total number of times the child was called.  Recursive calls by the child are not listed in the number after the `/'.

name   This is the name of the child. The child's index number is printed after it. If the child is a member of a cycle, the cycle number is printed between the name and the index number.

If there are any cycles (circles) in the call graph, there is an entry for the cycle-as-a-whole. This entry shows who called the cycle (as parents) and the members of the cycle (as children.) The `+' recursive calls entry shows the number of function calls that were internal to the cycle, and the calls entry for each member shows, for that member, how many times it was called from other members of the cycle.

Index by function name